Test-Case Generation for the Validation of Integrated Automation Systems Engineering Environments

Richard Mordinyi, Thomas Moser, Stefan Biffl

Christian Doppler Laboratory for Software Engineering Integration for Flexible Automation Systems
Vienna University of Technology – 1040 Wien, Favoritenstrasse 9-11/188 – Austria
{firstname.lastname}@tuwien.ac.at

Abstract: Automation Systems Engineering Projects typically involve the cooperation of a set of different engineering disciplines and therefore heavily rely on systems integration approaches. However, the configuration of the underlying technical integration platform is a complex and error-prone task requiring high manual effort checking whether the integration of an engineering tool is compatible with all running engineering processes and affecting engineering tools. Therefore, the outcome of the integration is only known at the end of the integration process. In this paper we describe a model-based concept which facilitates the derivation of test cases for introducing test methods at an early stage of the integration process by using ontologies. Based on an industrial scenario we discuss that the approach helps verify compatibility of the integration solution as knowledge regarding the overall system is captured.

Keywords: Test Generation, Software Tools, Knowledge Engineering, System Integration, Validation, Data Models.

1. INTRODUCTION

Nowadays large-scale engineering projects (such as e.g., power plants or car manufacturing plants) typically involve a broad range of engineering disciplines - as shown by Biffl et al. (2009), like mechanical, electrical or software engineering, which rely on their specific engineering tools and engineering systems to manage their specific engineering processes. For realizing and managing such complex projects a cooperation of all disciplines is required to form an integrated engineering system. However, error-prone and time-consuming human work is often needed to handle integration concerns at the interfaces of engineering disciplines and to configure the integration platform on the technical detail level, which is typically focused on a single technology or vendor. The configuration of such system integration technologies is a complex task requiring high manual effort and only doable by designated integration experts (IEs) as described in Hohpe and Woolf (2003) or with appropriate domain knowledge. Manual work is needed to write scripts for transformations and connection purposes with existing integration solution in case engineering tools are added to or updated in this solution, resulting in high efforts to check whether an engineering tool is compatible with running engineering processes. Consequently, as knowledge regarding the overall system is captured in non-machine understandable documents, the outcome of the integration is only known at the end of the integration process.

Biffl and Schatten (2009) proposed a platform called Engineering Service Bus (EngSB) which integrates not only different tools and systems but also different steps in the software development lifecycle. The platform introduces the concept of tool domains that provides interfaces for solving a common problem, independent of the vendor-specific tool used. The concept allows the EngSB to interact with a tool domain without knowing which specific tool instances are currently available, see Biffl et al. (2010). Tool domains do not implement tool instances but provide the abstract description of events and services, which have to be provided by concrete connectors of tool instances. In Moser et al. (2010), Moser et al. (2011), Mordinyi et al. (2011) a semi-automated system configuration process (MDSC) with focus on the EngSB has been introduced which helps system integrators to derive components and configuration artefacts (e.g., template source code artefacts, transformation instructions) using captured domain expert knowledge in ontologies. The captured knowledge describes data models and functionalities of tool instances, tool domains, and restrictions regarding data model elements.

This paper describes a model-based concept which facilitates the derivation of test cases for introducing test methods at an early stage of the integration process. It discusses the effectiveness of the intended integration solution by explaining how the captured knowledge may be used to verify intended integration solutions. First, the approach allows only those combinations of tool instances and tool domains for which the data models are transformable. Second, based on the ontology description of tool domains and tool instances the derived test cases verify the proper handling of data values as described in the data model, the proper handling of exceptions in case values are invalid or violate described restrictions, and the proper transformation between input and output data in the tool connector implementation.

The remainder of this paper is structured as follows: Section 2 summarizes related work, section 3 presents a motivating
use case, section 4 describes the solution approach, while section 5 discusses its advantages and limitations. Finally, section 6 concludes the paper and suggests further work.

2. RELATED WORK

This section summarizes related work on Automation Systems Engineering (ASE), Model-Driven Configuration and on Ontology-Based Generation of SE Artefacts.

2.1 Automation Systems Engineering

Automation systems engineering (ASE) projects typically involve heterogeneous environments, where engineers from different disciplines collaborate and interact with each other, as shown in Biffl et al. (2009). Major challenges arise from this need for collaboration across disciplines as individual engineers apply domain specific tools based on domain specific data models within domain specific and isolated environments, e.g., for constructing the mechanical layout (mechanical engineer), electrical circuit plans (electrical engineers), software models based on UML, and function plans for implementing functional control software (software engineers). Integration of engineering systems is a challenge as (particularly in the automation industry) typically a broad range of engineering tools from different vendors are used to solve specific problems, refer to Rangan et al. (2005). Tools within one vendor are sometimes integrated to exchange data, but hardly between vendors. APIs and exchange formats often do not follow established (open) standards. Therefore the AutomationML, presented in Schlepen and Drath (2009), project provides a standardized XML data exchange basis for data integration between multivendor automation systems engineering tools as foundation systematic information exchange between engineering models. The Engineering Service Bus (EngSB) described in Biffl et al. (2009), Biffl and Schatten (2009) provides a middleware platform that enables defect detection in overlapping areas of data models, i.e., common attributes of individual disciplines, based on common concepts in a virtual common data model, as shown in Moser and Biffl (2010). Semantically integrated data models can help detecting defects (a) in pre-engineering phases (i.e., project configuration and setup), (b) during the engineering phase (i.e., automation systems development in distributed heterogeneous environments), and (c) during run-time (i.e., linking run-time data to engineering objects for maintenance and condition monitoring purposes.

2.2 Model-Driven Configuration

The major goal of the Model Driven Architecture (MDA) approach is the separation of system functionality specification and implementation, as explained by Hohpe (2006). According to Halevy (2005) the advantages of the MDA framework are (1) automated generation of results improving productivity, development duration, and cost; (2) focusing on the creation of conceptual models rather than on logical and technical details. In contrast to the MDA approach, the Model-driven System Configuration (MDSC), presented by Mordinyi et al. (2009), Biffl et al. (2010), automatically derives integration technology configurations from business requirements rather than implementation code. Based on semantically described requirement and capability models which represent documents, integration expert knowledge and estimation/measurements of the integration network capabilities – see Hohpe and Woolf (2003), a logical solution model which represents the set of suitable integration partners (i.e. business services) is derived automatically, refer to IEEE (2007). The logical solution model is then transformed into a technical solution model that represents the specific integration configuration for the underlying integration network technologies. Additionally, it focuses on the communication sequence between tool instances based on a description of an engineering process facilitating the efficient integration of tool instances.

2.3 Ontology-Based Generation of SE Artefacts

The emerging field of semantic web technologies promises new stimulus for Software Engineering (SE) research. However, since the underlying concepts of the semantic web have a long tradition in the knowledge engineering field, it is sometimes hard for software engineers to overlook the variety of ontology-enabled approaches to Software Engineering. The work of Zhao et al. (2009) serves as a starting point to elaborate a systematic categorization of the approaches and to derive more clearly defined acronyms. In the context of this paper, the usage of ontologies can be specified as Object-Oriented Design Ontology, Source-Code Ontology and Testing Ontology types. Happel and Seedorf (2006) furthermore distinguish the role of ontologies in the context of Software Engineering between usage at run time and at development time, and additionally look at the kind of knowledge the ontology actually compromises. Here, they distinguish between the problem domain that the software system tries to tackle, and infrastructure aspects to improve the software or its development. In the context of this paper, ontologies can be used as flexible data models for system and test definition.

Software tests are an important part of quality assurance, as described by Abran et al. (2001). However, writing test cases is an expensive endeavour that does not directly yield business value. It is also not a trivial task, since the derivation of suitable test cases demands a certain amount of domain knowledge. Ontologies could help generate basic test cases since they encode domain knowledge in a machine-understandable format. A simple example for this would be regarding cardinality constraints. Since those constraints define restrictions on the association of certain classes, they can be used to derive equivalency classes for testing, see Knublauch et al. (2006). Ontologies may not be the first candidate for such a scenario, since there are formalisms like OCL1 that are specialized for such tasks. However, once domain knowledge is available in an ontology format, it might be feasible to reuse that knowledge.

Nguyen et al. (2008) describe a framework for automated test case generation in the context of multi-agent systems. They use agent interaction ontologies that define content semantics

1 http://www.omg.org/spec/OCL/2.3.1/
of agent interactions to generate test inputs, guide the exploration of the input space during test case generation, and verify messages exchanged between agents with respect to the agent interaction ontology. Their results for interaction ontologies of non-trivial size show that the ontology-based method achieves higher coverage of the ontology classes than manual test case derivation. The ontology-based approach also outperformed manual derivation in terms of defects detected and coverage of input space.

3. MOTIVATING SCENARIO

The industrial scenario has been retrieved from an industrial partner developing, creating, and maintaining hydro power plants. One of the base artifacts in the course of developing power plants are signals which consist of structured key value pairs created by different hardware components. Today’s integrated tool suites often consist of a pre-defined set of tools and a homogeneous common data model, which work well in their narrow scope but do not easily extend to other tools in the project outside the tool’s scope. However, depending on the size of the commissioned power planted there are about 40 to 80 thousand signals to be managed and administrated in different tools of different engineering disciplines. Therefore, system integrators in multi-disciplinary engineering projects want to be able to conduct automated change management across all tools that contribute project-level data elements regardless of the origin of the tool and data model.

While there exist some engineering processes which handle contributions of engineers from different engineering disciplines as a sequence of steps, in practice engineers tend to concurrently update their artifacts (e.g., documents, plans) originating from different tools in the engineering process, to address new requirements or issues. However, the possibility of parallel updates on planning data of distributed domains requires the correct integration of engineering tools into existing integration solutions without effecting current engineering processes or introducing invalid data updates. Therefore, validation of an integration solution is necessary before deploying it.

4. SOLUTION APPROACH

This section describes a model-based concept which facilitates the derivation of test cases for introducing test methods at an early stage of the integration process. According to Moser et al. (2010), Moser et al. (2011) system integrators capture domain expert knowledge in ontologies which is structured into several levels as presented in Moser et al. (2009): a) the tool instance level describes the data models and restrictions placed on the data of specific engineering tools; b) the tool domain level describes the common concepts certain tools have in common and provides an additional data models; and finally c) the application domain level captures concepts which are relevant for entire engineering processes and common for similar scenarios of the same application domain. In overall, the captured knowledge describes data models and functionalities of tool instances, tool domains, and restrictions regarding data model elements.

Based on these captured information, system integrators are capable of deriving configurations and source code artefacts to create an overall technical integration solution by using a semi-automated model-driven configuration approach, described in Mordinyi et al. (2011). As shown in Figure 1, the process derives (1) tool domain interfaces and method calls; (2) tool connector source code templates with implementations of checks to verify whether all restrictions regarding the described data elements are fulfilled; (3) data model transformation instructions to translate data models of tool instances to the data model of the tool domain and between data models of tool domains; and (4) tests cases derived for tool instances by deriving test cases using the signature (e.g., method name, parameters, parameter types) of the tool domain. The following paragraphs describe the derived artefacts in more details:

(1) Tool Domain Interfaces. The tool domain ontologies capture the method signatures of the methods provided by a specific tool domain. This includes the identifying name of the method, a set of zero or more method parameters consisting of a method parameter type (e.g., integer or string) and a method parameter name, and the return value of the method (either void if the message does not return any value or the type and name of the return value similar to the method parameter definition). Based on this information, the tool domain interface can be generated. This interface provides an abstract definition of the tool domain functionality and consists of all the methods specified in the tool domain ontology describing the specific tool domain.

The example shown in the top left hand side of Figure 1 describes the Signal Domain. This domain defines two methods, namely SaveSignal and LoadSignal. The SaveSignal method specifies three method parameters, Signal ID, Signal Source and Signal Description, and has its message return value defined as void. The LoadSignal method specifies a single message parameter, Signal ID, and has its message return value defined as type Signal using the name returnSignal.

(2) Tool Connector Stub Implementation. Analogous to the tool domain ontology, each single engineering tool is modelled in the tool instance ontology. Again, the tool instance method signatures are described as explained in the previous paragraphs. Furthermore, it is possible to specify restrictions for each of the method parameters. More details on these restrictions can be found in (3). The mapping ontology provides the glue between the tool instance and tool domain ontologies. For each of the tool domain methods defined in the tool domain ontology, the mapping ontology provides a link to the corresponding method description in the tool instance ontology, as usually the method names are not equal. Furthermore, as the parameters used for method descriptions on tool domain and tool instance level may differ both in number and types of method parameters used, also the mappings between the method parameters (which can also be seen as single attributes of tool instance or tool domain data model entities) are specified in the mapping ontology. These mappings are primarily used for deriving transformation instructions as described in (4).
Finally, the mapping ontology contains mappings between the method return values on tool domain and tool instance levels, which are defined similar as the mappings between tool domain and tool instance method parameters. Based on this information, a stub implementation of the tool connector can be generated. This tool connector stub implements the particular tool domain interface of the tool domain the tool instance belongs too. For each of the tool domain methods inherited from the tool domain interface, the tool connector stub contains the call to the method defined by the tool instance. If transformation of the method parameters is required, the specific transformation instructions are called prior to the tool instance message call.

The example presented in the middle and right hand side of Figure 1 shows the mappings between two tool instances, EPlan and OPM, and the Signal Domain they belong to. E.g., OPM specifies the method Save_Variable which is mapped to the Save_Signal method defined in the Signal Domain. For the method call, the parameters of OPM have to be transformed to comply with the Signal Domain data model: region will be transformed directly to Signal_ID, cpuNumber will be transformed directly to Signal_Source, and func will be transformed to Signal_Description by replacing the character at index 1 with the character at index 3. More details on the supported basic transformation operations can be found in (4).

(3) Restrictions for Stub Implementation of Tool Connector. As described in the previous section, it is possible to define restrictions regarding specific tool instance method parameters (tool instance data model attributes) in the tool instance ontology. By now, the prototypic implementation supports the modelling and generation of basic and extended restrictions. As shown in the top right hand side of Figure 1, basic restrictions include integer comparisons (e.g., OPM.region has to be smaller than 254 or OPM.cpuNumber has to be smaller than 20) and regular expressions for checking text-based values (e.g., EPlan.Source has to consist of exactly 6 characters). Extended restrictions, which are also known as plausibility checks, are principally a set of restrictions that has to be fulfilled by a set of tool instance data model attributes at the same time (e.g., if parameter A is greater 10 and parameter B is “AAA”, then parameter C has to be 0). Such checks often encode domain-specific knowledge and therefore are hard to generalize in an ontological model, however the architecture of our tool instance ontology allows for the definition of user-defined restrictions, which can be generated into the tool connector stub using a domain-specific generator.

The explicit specification of restrictions regarding specific tool instance method parameters allows the automated generation of JUnit test cases for the stub implementations of tool connectors (see the bottom right hand side of Figure 1). Since both the type of the method parameters, as well as the restricted values for these parameters are explicitly modeled in the tool instance ontologies, method calls can be generated automatically to test the generated tool connector stub, either as positive test cases (i.e., all generated method parameters comply with the modeled restrictions) or as negative test cases (i.e., some generated method parameters violate some of the modeled restrictions).

(4) Transformation Instructions. As described in (2), some tool instance data model entity attributes have to be transformed in order to allow a method call on tool domain level. This may originate e.g., from different naming conventions or

\[ \text{http://www.junit.org/} \]
standards, but also from different granularity levels of the particular data models. As shown in the top middle of Figure 1, for each target field, all required source fields are explicitly listed in the mapping ontology. Furthermore, the mapping ontology contains basic transformation operations (in the prototypic implementation three basic transformation operations are included: split of a message parameter, merge of a set of message parameters and the replacement of one or more characters of a message parameter), which can be modelled in the mapping ontology and also generated as transformation instructions. In addition, advanced transformation operations requiring user input can be modelled as a template specifying the transformation input and output, which has to be implemented by the user prior to the generation of the transformation instructions.

Based on the defined input and output data model entity attributes, test cases for testing specific transformation instructions can be generated automatically (see the bottom right hand side of Figure 1). Since the types of input and output are explicitly specified in the tool domain and tool instance ontologies, exemplary data can be generated and fed into a transformation instruction. This allows to test whether the set of modelled and implemented transformation operations work as intended and also provides valuable feedback to the user. In the prototypic implementation, we provide this feedback directly next to the model of a transformation instruction, allowing the user to instantly test every change made to the transformation instruction model.

Finally, the combination of the tests of the transformation instructions and of the tool instance method parameter restrictions can be combined to represent overall JUnit tests for validating the communication between tool instances and EngSB (as shown in the bottom right side of Figure 1). This allows to test, and therefore to validate, the connection of single engineering tools to the EngSB. Each of the methods defined by the tool domain and implemented in the tool connector stub can be tested regarding the validity and functionality of the method parameter transformation compliance on the one hand side, and on the other hand side the compliance of the method parameters with defined restrictions.

5. DISCUSSION

The effectiveness of the intended integration solution is verified in two different ways. First, the model-driven approach allows only those combinations of tool instances and tool domains for which the data models are transformable. Second, based on the ontology description of tool domains and tool instances the derived test cases verify the proper handling of data values as described in the data model, the proper handling of exceptions in case values are invalid or violate described restrictions, and the proper transformation between input and output data in the tool connector implementation.

Compared to standard transformation approaches, such as the ATL3 (ATLAS Transformation Language), the proposed method provides a more domain-specific and therefore more intuitive approach to defining transformations. ATL is an Eclipse based model transformation language that provides ways to produce a set of target models from a set of source models. It originally has been initiated by OMG and is part of Eclipse's M2M (Model to Model transformation) project. The ATL approach is noble, though transformation rules are defined in a single, transparent file. ATL supports syntax highlighting and debugging. Transformation rules are intuitive to interpret and may be easier to read than XSLT, because the rules do not contain any XML. Helper functions can be defined and allow more freedom to the developer. ATL is easy adaptable. Drawbacks are, that the Eclipse environment is hard to setup. Some of the required components are still in incubation phase and incompatible either among themselves or with some versions of Eclipse. Furthermore, ATL training effort is high. Developers have to dive into the EMF concept to gain a deeper understanding and learn ATL, which can be time consuming. In the proposed method, also the separation between the modelling and the implementation of transformations allows for an easier reuse and faster refactoring of available transformations. By now the prototypic implementation supports a set of basic operations (split, merge, replace) as well as a method to model input/output templates for user-defined advanced transformation operations as described in the previous section.

We initially evaluated the proposed concepts in an industrial use case in the context of manufacturing hydro power plants. First results show that the effort for the initial configuration of the integration solution is higher due to the need to capture domain expert knowledge in ontologies in comparison to manual configuration. However, in case of adaptations the effort is significantly lower as updates effects only the models. Furthermore, the validation of the integration solution is already performed during modelling and through the derived test cases executed immediately after the execution of the model-driven system configuration process. Nevertheless, the derived test cases are only capable to check the correct handling of data based on provided input and retrieved output values, rather than based on the real semantics of the method.

In the prototypic implementation, two types of test cases are generated, script-based test cases for transformation instructions and JUnit test cases for the tool connector stub implementations. As described in the previous section, these two types can be combined in order to test and therefore to validate the communication between single engineering tool instances and the EngSB. For generating the test cases for the transformation instructions, the modelled types of input and output data model entry attributes are used as basis, while for generating the test cases for the tool connector stub instances the defined restrictions are used as basis to generate both positive and negative test cases.

6. CONCLUSIONS

In typical Automation Systems Engineering Projects the cooperation between different engineering disciplines is required for which sophisticated systems integration approaches are essential. Using integration knowledge captured in ontologies, in this paper we introduced a model-based approach that facilitates the derivation of test cases help verify
an integration solution at an early stage of the integration process. Derived test cases are capable of testing transformation instructions and communication adapters, both used to exchange data between heterogeneous engineering tools. The concept is however limited to test syntactical aspects of an implementation (e.g., interfaces, parameters, restrictions) rather than the semantics of behaviour of the implementation. For future work the modelling of plausibility checks in the used ontologies is intended which may help increase the effectiveness of the approach.

REFERENCES


